Ruby

Ruby Hacking Guide

ソースコード完全解説

青木 峰郎 著／まつもとゆきひろ 監修

インプレス

目錄

Ruby 源码解读

> 原文：Rubyソースコード完全解説
>
> 译者：gyorou
>
> 来源：试译 Ruby 源码解读

(申明：仅供学习参考，本人不负任何责任，如有违规请联系删除)

第一章 最低限度的ruby知识

为了方便第一部分的解说，在这里简单介绍一下ruby的基本知识。这里不会系统介绍编程的技巧方面的东西，读完这章节也不会让你掌握ruby的编程方法。如果读者已经有ruby的经验，那就可以直接跳过本章节。

另外我们会在第二部分不厌其烦地讲解语法，于是在这一章节我们尽量不涉及语法相关内容。关于hash，literal之类的表示方法我们会采用经常使用的方式。可以省略的东西原则上我们不会省略。这样才会使语法看起来简单，但是我们不会重复提醒"这里可以省略"。

对象

字符串

ruby程序造作的所有东西都是对象。在ruby里面没有如java里面的 int 或者 long一样的基本数据类型。比如，如下面的例子一样书写的话，就会生成一个内容是"content"的字符串(String)对象。

```
"content"
```

刚才说这个只是个字符串的对象，其实正确来讲这个是生成字符串对象的表达式。所以每写一次，都会有新的字符串对象生成。

```
"content"
"content"
"content"
```

这里就生成了三个内容是"content"的对象。

但是单纯有对象程序是看不到的。下面教你如何在终端显示对象。

```
p("content") #显示"content"
```

"#"之后的东西是注释。接下这个都表示注释。

"p(……)"是调用了函数p。它能够较逼真的表示对象的状态，基本上是一个debug用的函数。

虽然严格意义上讲，ruby里面并不存在函数这个概念。但是现在请先把它当作函数来理解。函数无论在哪里都可以使用。

各种各样的序列

接下来我们来说明一下直接生成对象的序列(literal)。先从一般的整数和小数说起。

```
# 整数
1
2
100
9999999999999999999999999   # 无论多大的数都可以使用

# 小数
1.0
99.999
1.3e4     # 1.3×10^4
```

请记住，这些也全部是生成对象的表达式。在这里我们不厌其烦得重复强调，ruby里面是没有基本类型的。

下面是生成数组对象的表达式。

```
[1,2,3]
```

这个程序会按顺序生成包含1 2 3 三个元素的数组。数组的元素可以是任何对象。于是也可以有这种表达式。

```
[1, "string", 2, ["nested", "array"]]
```

甚至，下面的用法可以来生成哈希表。

```
{"key"=>"value",  "key2"=>"value2",  "key3"=>"value3"}
```

所谓哈希表，是指任何的所有对象之间的一对一的数据结构。按照上述写法会构造出包含下面所示关系的一张表。

```
"key"   →  "value"
"key2"  →  "value2"
"key3"  →  "value3"
```

这样生成了哈希表之后，当我们向此对象询问"key对应的值是什么"的时候，它就会告诉我们结果是"value"。那该怎么询问呢？那就要用到方法了。

方法的调用

对象可以调用方法。用c++的话说就是调用成员函数。至于什么是方法，我觉得没有必要说明，还是看一下下面这个简单的例子吧。

```
"content".upcase()
```

这里调用了字符串(字符串的内容为"content")的upcase方法。upcase是返回一个将所有小写字母全部变成大写的字符串。于是会有下面的效果。

```
p("content".upcase())   # 输出"CONTENT"
```

方法可以连续调用

```
"content".upcase().downcase()
```

这个时候调用的是"content".upcase()的返回值对象的downcase方法。

另外，ruby里面没有像Java和C++一样的全局概念。于是所有对象的接口都是通过方法来实现。

程序

顶层

在ruby里面写上一个式子就已经算是一个程序了。没有像C++或者Java一样需要定义main()。

```
p("content")
```

仅仅就这样已经算是一个完整的ruby程序。把这个东西复制到first.rb这个文件中然后在命令行执行

```
% ruby first.rb
"content"
```

使用选项-e可以不需要创建文件就可以执行代码。

```
% ruby -e 'p("content")'
"content"
```

然而要注意到的是，上面p的位置是程序中最外层的东西，也就是说在程序上属于最上层，于是被称作顶层。有层顶是ruby脚本语言的最大特征。

ruby基本上一行就是一句。不需要在句尾加上分号。所以下面的内容实际上是三个语句。

```
p("content")
p("content".upcase())
p("CONTENT".downcase())
```

执行结果如下

```
% ruby second.rb
"content"
"CONTENT"
"content"
```

局部变量

ruby中所有的变量和常量都仅仅是对对象的引用(reference)。所以仅仅是带入其他变量的话并不会发生复制之类的行为。 这个可以联想到Java中的对象型变量，以及C++中的指向对象的指针。但是指针的值无法改变。

ruby仅看变量的首字母就可以分别出变量的种类。小写阿拉伯字母或者下划线开始的变量属于局部变量。使用等于号=代入赋值。

```
str = "content"
arr = [1,2,3]
```

一开始代入的时候不需要声明变量，另外无论变量是什么类型，代入方法都没有区别。下面的写法都是合法的。

```
lvar = "content"
lvar = [1,2,3]
lvar = 1
```

当然，虽然可以这么写但是完全没有必要故意写成这样。把种类各样的对象放在一个变量中会使得代码变得晦涩难懂。现实中很少有如此的写法，在这里我们仅仅是举个例子。

变量内容查询也是常用的。

```
str = "content"
p(str)           # 结果显示"content"
```

下面我们从变量持有对象的引用的观点来看下面的例子。

```
a = "content"
b = a
c = b
```

执行这个程序之后，变量a，b，c三个局部变量指向的都是同一个对象，也就是第一行生成的"content"这个字符串对象。

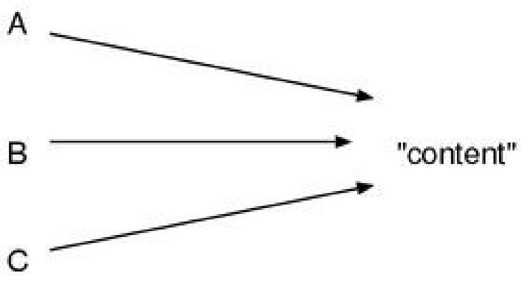


图1: Ruby的变量拥有对对象的引用

这里我们注意到，一直在说的局部变量，那么这个局部必然是针对某个范围的局部。但是请稍等片刻我们再做解释。总之我们先可以说顶层也是一个局部的作用域。

常量

名称首字母是大写的称作常量。所谓常量就是只能代入赋值一次。

```
Const = "content"
PI = 3.1415926535

p(Const)   # 输出"content"
```

带入两次的话会发生错误。虽然按道理是这样，但是实际运行的时候却不会有错误例外发生。这个是为了保证执行ruby程序的应用程序，比如说在同一个开发环境下，读入两个同样的文件的时候不至于产生错误。也就是说是为了实用性而不得不做出的牺牲，万不得已才取消了错误提示。实际上在Ruby1.1之前是会报错的。

```
C = 1
C = 2   # 实际上只会给出警告，最理想的还是要做成会显示错误
```

接下来，被常量这个称呼欺骗的人肯定有很多。常量指的是"一旦保存了所要指向的对象就不再会改变"这个意思。而常量指向的对象并不是不会发生变化。如果用英语来说，比起constant这个意思，还是read only来的比较贴切。(图2)。顺带一提，如果要使对象自身不发生变化可以用freeze这个方法来实现。

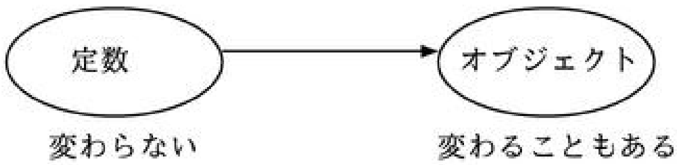


图2： 常量是read only的意思

另外这里还没有提到常量的作用域。我们会在下一节的类的话题中来说明。

流程控制

Ruby的流程控制结构很多要列举的话举不胜举。总之就介绍一下基本的if和while。

```
if i < 10 then
  # 内容
end

while i < 10 do
  # 内容
end
```

在条件语句中只有false和nil两个对象是假，其他所有对象的都属于真。当然0和空字符串也是真。

顺带一提，只有false的话感觉不怎么美观，于是当然也有true，当然true是属于真。

最纯粹的面向对象的系统中，方法都是对象的附属物。但是那毕竟只是一个理想。在普通的程序中会有大量的拥有相同方法集合的对象。如果还傻傻的以对象为单位来调用方法那就会造成内存的大大的浪费。于是一般的方法是使用类或者multi-method来避免重复定义。

ruby采用了传统上来连接方法和对象的结构，类。也就是所有的对象都必须属于唯一的一个类，这个对象能够调用的方法也友这个类来决定。这个时候对象通常被叫做“某某类的实例”。

例如"str"就是类String的实例。另外，String这个类还定义了upcase，downcase，strip等一系列其他方法，仿佛就是在说所有的字符串对象都可以利用这些方法一样。

```
# 大家都属于同一个String类于是拥有相同的方法。
       "content".upcase()
"This is a pen.".upcase()
    "chapter II".upcase()

       "content".length()
"This is a pen.".length()
    "chapter II".length()
```

那么，如果调用的方法没有事前定义会发生什么呢？在静态的语言中，编译器会报错，而在ruby中，执行的时候会抛出例外。我们来实际尝试一下。这点长度的话直接在命令行用-e就好了。

```
% ruby -e '"str".bad_method()'
-e:1: undefined method `bad_method' for "str":String (NoMethodError)
```

找不到方法的时候会发生NomethodError这个错误。

另外最后，每次都说一遍类似“String的upcase方法”太烦了，于是我们下面就用“String#upcase”来表示“在String的类中定义好的upcase方法”。

顺带一提，如果写成“String.upcase”在ruby中就又是另一个意思了。

类的定义

到目前为止都是针对已经定义好的类。当然我们也可以自己来定义类。定义类的时候使用class关键字。

```
class C
end
```

这样就定义了C这个类。定义好之后可以有以下的使用。

```
class C
end
c = C.new()   # 生成类C的实例带入c中。
```

注意生成实例的写法不是 `new C` 。恩，好像 `C.new` 这个写法在调用方法一样呢。如果你这样想，那说明你很聪明。Ruby生成对象的时候仅仅是调用了方法而已。

首先在Ruby里面类名和常量名是一个概念。那么类名和同名的常量的内容到底是什么呢？实际上里面是类。在Ruby中一切都是对象，那么当然类也是对象了。我们姑且将其称作类对象。所有的类对象都是Class这个类的实例。

也就是说class这个写法，完成的是生成新的类对象的实例，并且将类名带入和它同名的常量中这一系列操作。同时，生成实例，实际上是参照常量名，对该类对象调用方法(通常是new方法)的操作。看完下面的例子你就应该会明白生成实例和普通的调用方法根本就是一回事。

```
S = "content"
class C
end

S.upcase()  # 获取常量S所指的对象并调用upcase方法。
C.new()     # 获取常量C所指的对象并调用new方法。
```

因此，在Ruby里面new并不是保留词。 另外我们也可以用p来显示刚刚生成的类的实例。

```
class C
end

c = C.new()
p(c)       # #<C:0x2acbd7e4>
```

当然不可能像字符串和整数那样漂亮地显示出来，显示出来的是类内部的ID。顺带一提这个ID其实是指向对象的指针的值。

对了。差点忘了说明一下方法名的写法。 `Object.new` 是类对象Object调用自己的方法new的意思。 `Object#new` 和 `Object.new` 完全是两码事情，必须严格区分。

```
obj = Object.new()   # Object.new
obj.new()            # Object#new
```

在刚刚的例子中因为还没定义 `Object#new` 方法所以第二行会报错。我们就当是一个写法的例子来理解就好了。

方法定义

就算定义了类如果不定义方法的话一般没有什么太大意义。我们继续定义类C的方法。

```
class C
  def myupcase( str )
    return str.upcase()
  end
end
```

定义类用的是def。这个例子中定义了myupcase这个方法。这个方法带一个参数str。和变量同样，没有必要写明返回值的类型和参数的类型。另外参数的个数是没有限制的。

我们来使用一下定义好的方法。方法默认可以从外部调用。

```
c = C.new()
result = c.myupcase("content")
p(result)   # 输出"CONTENT"
```

当然习惯了之后也没必要每次都带入。下面的写法也是同样的效果。

```
p(C.new().myupcase("content"))   # 同样输出"CONTENT"
```

self

在执行方法的过程中经常要保存自己是谁(调用方法的实例)这个信息。self可以取得这个信息。在C++或者Java中就是this。我们来确认一下。

```
class C
  def get_self()
    return self
  end
end

c = C.new()
p(c)              # #<C:0x40274e44>
p(c.get_self())   # #<C:0x40274e44>
```

如上所示，两个语句返回的是用一个对象。也就是说对实例c调用方法的时候slef就是c自身。

那么要如何才能对自身的方法进行调用呢？首先可以考虑通过self。

```
class C
  def my_p( obj )
    self.real_my_p(obj)   # 调用自身的方法。
  end

  def real_my_p( obj )
    p(obj)
  end
end

C.new().my_p(1)   # 输出1
```

但是调用自身的方法每次都要这么表示一下实在是太麻烦。于是可以省略掉self，调用自身的方法的时候直接就可以省略掉调用方法的对象(receiver)。

```
class C
  def my_p( obj )
    real_my_p(obj)   # 无需指定receiver
  end

  def real_my_p( obj )
    p(obj)
  end
end

C.new().my_p(1)   # 输出1
```

实例变量

对象的实质就是数据+代码。这个说法表明，仅仅定义了方法还是不够的。我们需要以对象为单位来保存数据。也就是说我们需要实例变量。在C++里面就是成员变量。

Ruby的变量命名规则很简单，是由第一个字符决定的。实例变量以@开头。

```
class C
  def set_i(value)
    @i = value
  end

  def get_i()
    return @i
  end
end

c = C.new()
c.set_i("ok")
p(c.get_i())   # 显示"ok"
```

实例变量和之前的所介绍的变量稍微有点区别，不用代入(也无需定义)也可以可以引用。这个时候会是什么情形呢……我们在之前代码的基础上来试试看。

```
c = C.new()
p(c.get_i())   # 显示nil
```

在没有set的情况下调用get，会显示nil。nil是表示什么都没有的对象。明明自身是个对象却表示什么都没有这个的确有点奇怪，但是它就是这么一个玩意儿。

nil也可以用序列来表示。

```
p(nil)   # 显示nil
```

初始化

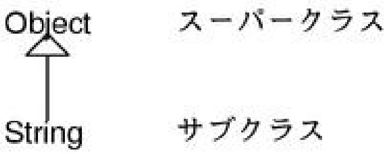
到目前为止，正如所见，就算是刚定义好的类只要调用new方法就可以生成实例。的确是这样，但是有时候类也需要特殊的初始化吧。这个时候我们就不是去改变new，而是应该定义initialize这个方法。这样的话，在new中就会调用这个方法。

```
class C
  def initialize()
    @i = "ok"
  end
  def get_i()
    return @i
  end
end
c = C.new()
p(c.get_i())   # 显示"ok"
```

严格意义上来说这个初始化只是new这个方法的特殊设计而不是语言层次上的特殊设计。

继承

类可以继承其他类。比如说String类就是继承的Object这个类。在本书中将用下图的箭头来表示。



上图的情况下，被继承的类(Object)叫做父类或者上层类，继承的类(String)叫做下层类或者子类。注意这个叫法和C++有所区别。但和Java是一样的喊法。

总之我们来尝试一下，我们来定义一个继承别的类的类。要定义一个继承其他类的类的时候有以下写法。

```
class C < SuperClassName
end
```

到目前为止，我们凡是没有标明父类的类的定义，其实继承的都是Object这个父类。

接下来我们考虑一下为什么要继承，当然继承是为了继承方法了。所谓继承，仿佛就是再次重复了一下在父类中第一的方法。

```
class C
  def hello()
    return "hello"
  end
end

class Sub < C
end

sub = Sub.new()
p(sub.hello())   # 输出"hello"
```

虽然hello方法是在类C中定义的，但是Sub这个类的实例可以调用这个方法。当然这次不需要带入变量。下面的写法也是同样的效果。

```
p(Sub.new().hello())
```

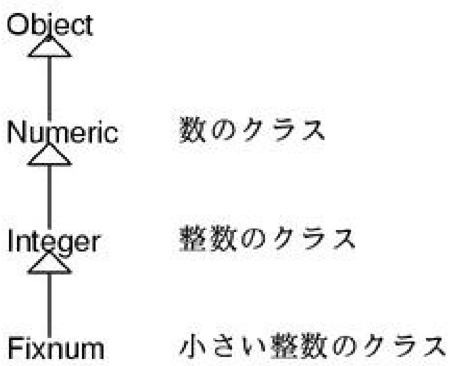
只要在子类中定义相同名字的方法就可以重载。在C++,Object Pascal(Delphi)使用virtual之类的保留字明示除了指定的方法之外无法重载，而在Ruby中所有的方法都可以无条件地重载。

```
class C
  def hello()
    return "Hello"
  end
end

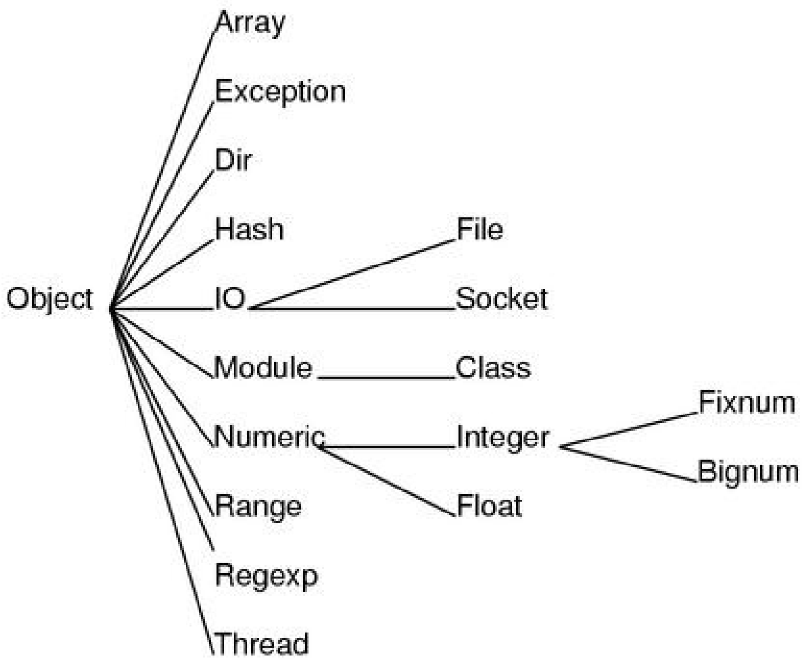
class Sub < C
  def hello()
    return "Hello from Sub"
  end
end

p(Sub.new().hello())   # 显示"Hello from Sub"
p(C.new().hello())     # 显示"Hello"
```

另外类可以多层次继承。如图4所示。这个时候Fixnum继承了Object和Numeric以及Integer的所有方法。如果有同名的方法则以最近的类的方法为优先。不存在因类型不同而发生的重载所以使用条件非常简单。



另外C++中可以定义没有继承任何类的类，但是Ruby的类必然是直接或者间接继承Object类的。也就是说继承关系是以Object在最顶端的一棵树。比如说，基本库中的重要的类的继承关系树如图5所示。



父类被定义之后绝对不会被改变，也就是说在类树中添加新的类的时候，类的位置不会发生变化也不会被删除。

变量的继承……？

Ruby中不继承变量(实例变量)。因为就算想继承，也无法获取类中使用的变量的情报。

但是只要是继承了方法那么调用继承的方法的时候(以子类的实例)会发生实例变量的带入。也就是说会被定义。这样的话实例变来那个的命名空间在各个实例中是完全平坦的，无论是哪个类的方法都可以获取。

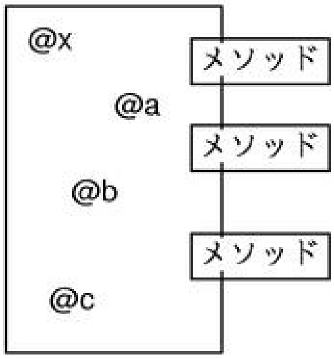
```
class A
  def initialize()   # 在new的时候被调用
    @i = "ok"
  end
end

class B < A
  def print_i()
    p(@i)
  end
end

B.new().print_i()   # 显示"ok"
```

如果无法理解这个行为那么干脆别去考虑类和继承。就想象一下如果类C存在实例obj，那么C的父类的所有方法都在C中有定义。当然也要考虑到重载的罪责。然后现在把C的方法和obj衔接在一起。(图6)

这种强烈的真实感正是Ruby的面向对象的特征。



模块

父类只能指定一个，也就是说Ruby表面上是单一继承的。实际上因为存在模块所以Ruby拥有多重继承的能力。下面我们就来介绍一下模块。

```
module M
end
```

这样就定义了模块。在模块里定义方法和类中定义完全一样。

```
module M
  def myupcase( str )
    return str.upcase()
  end
end
```

但是因为模块无法生成对象所以是无法直接调用模块里面定义的方法的。那么该怎么办？恩将模块包含进其他的类里面就是了。这样的话模块就仿佛继承了类一样可以被操作了。

```
module M
  def myupcase( str )
    return str.upcase()
  end
end

class C
  include M
end

p(C.new().myupcase("content"))  # 显示"CONTENT"
```

虽然类C根本没有定义任何方法但是可以调用myupcase这个方法。也就是说它继承了模块M的方法。包含的机能和继承完全是一样的，无论是定义方法还是获取实例变量都不受限制。

模块无法指定父类。但是却可以包含其它模块。

```
module M
end

module M2
  include M
end
```

也就是说这个机能去其实和指定父类是一样的。但是模块上边是不会有类的，模块可以包含的只能是模块。

下面的例子包含了方法继承。

```
module OneMore
  def method_OneMore()
    p("OneMore")
  end
end

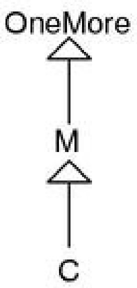
module M
  include OneMore

  def method_M()
    p("M")
  end
end

class C
  include M
end

C.new().method_M()         # 输出"M"
C.new().method_OneMore()   # 输出"OneMore"
```

Module也可以和类一样的继承图。



话说类C如果本身已经存在父类，那么这下子和模块的关系会变成什么样呢？请试着考虑下面的例子。

```
# modcls.rb

class Cls
  def test()
    return "class"
  end
end

module Mod
  def test()
    return "module"
  end
end

class C < Cls
  include Mod
end

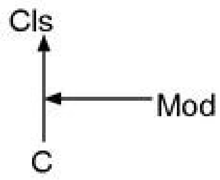
p(B.new().test())   # "class"？ "module"？
```

类C继承了类Cls，并且引入了Mod模块。这个时候应该表示"class"呢还是表示"module"呢？换句话说，模块和类哪一个距离更近呢？Ruby的一切都可以向Ruby询问。于是我们来执行看一下结果。

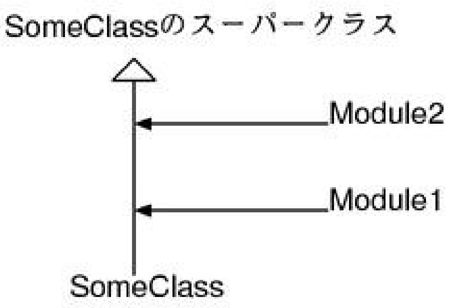
```
% ruby modcls.rb
"module"
```

比起父类模块的优先度更高呢！

一般来说，在Ruby中引入模块的话，会在类和父类之间夹带产生一个继承关系，如下图所示。



在模块中又引入模块的话会有以下的关系。



程序(2)

注意，在本节中出现的内容非常重要，并且主要讲的是一些习惯静态语言思维的人很难习惯的部分。其他的内容可以一眼扫过，而这里请格外注意。我会比较详细地说明。

常量的嵌套

首先复习一下常量。以大写字母开头的是常量。可以以如下方法定义。

```
Const = 3
```

要调用这个参数的时候可以如下。

```
p(Const)   # 输出3
```

其实写成这样也可以。

```
p(::Const)   # 同样输出3
```

在开头添加 `::` 表示是在顶层定义的常量。你可以类比一下文件系统的路径的表示方法。在root目录下有vmunix这个文件。如果是在"/"目录下的话直接输入vmunix就可以获取文件。当然也可以用完整的路径"/vmunix"来表示。`Const` 和 `::Const` 也是同样道理。如果是在顶层的话直接用 `Const` 就ok，当然使用完整路径版本的 `::Const` 也是可以的。

那么类比文件系统目录的东西，在Ruby中是什么呢？那就是类的定义和模块的定义了。因为每次都这么说一遍的话太烦了于是下面统一就用类的定义语句来代替。在类的定义语句中常量的等级会随之上升。(类比进入文件系统的目录)

```
class SomeClass
  Const = 3
end

p(::SomeClass::Const)   # 输出3
p(  SomeClass::Const)   # 同样输出3
```

SomeClass是在顶层定义的类，也是常量。于是写成 `SomeClass` 和 `::SomeClass` 都可以。在其中签到的常量 `Const` 的路径就变成了 `::SomeClass::Const`

就像在目录中可以继续创建目录一样，在类中也可以继续定义类。例如

```
class C         # ::C
  class C2      # ::C::C2
    class C3    # ::C::C2::C3
    end
  end
end
```

那么问题是，在类定义语句中定义的常量就必须要用完整的路径表示吗？当然不是了。就像文件系统一样，只要在同一层的类定义语句中，就不需要 `::` 来获取。如下所示。

```
class SomeClass
  Const = 3
  p(Const)   # 输出3
end
```

也许你会感到奇怪。居然在类的定义语句中可以直接书写执行语句。这个也是习惯动态语言的人相当不习惯的部分了。作者当初也是大吃一惊。

姑且再补充说明一点，在方法中也可以获取常量。获取的规则和在类的定义语句中(方法之外)是一样的。

```
class C
  Const = "ok"
  def test()
    p(Const)
  end
end

C.new().test()   # 输出"ok"
```

全部执行

这里从整体出发来写一段代码吧。Ruby中程序的大部分都是会被执行的。常量定义，类定义语句，方法定义语句以及其他几乎所有的东西都会以你看到的顺序依次执行。

例如，请看下面的代码。这段代码使用了目前为止说到的很多结构。

```
 1:  p("first")
 2:
 3:  class C < Object
 4:    Const = "in C"
 5:
 6:    p(Const)
 7:
 8:    def myupcase(str)
 9:       return str.upcase()
10:    end
11:  end
12:
13:  p(C.new().myupcase("content"))
```

这个代码将会按照以下的顺序执行。

1: p("first") 输出"first"

3: < Object 通过常量Object获取Object类对象。

3: class C 生成以Object为父类的新的类对象，并将其代入C

4: Const = "in C" 定义::C::Const。值为"in C"

6: p(Const) 输出::C::Const的值。输出结果为"in C"。

8: def myupcase(...)...end 定义方法C#myupcase。

13: C.new().myupcase(...) 通过常量C调用其new方法，并且调用其结果的myupcase方法。

9: return str.upcase() 返回"CONTENT"。

13: p(...) 输出"CONTENT"。

局部常量的作用域

终于说道了局部变量的作用域。

顶层，类定义语句内部，模块定义语句内部，方法自身都各自拥有完全独立的局部变量作用域。也就是说在下面的程序中lvar这个变量都不一样，也没有相互往来。

```
lvar = 'toplevel'

class C
  lvar = 'in C'
  def method()
    lvar = 'in C#method'
  end
end

p(lvar)   # 输出"toplevel"

module M
  lvar = 'in M'
end

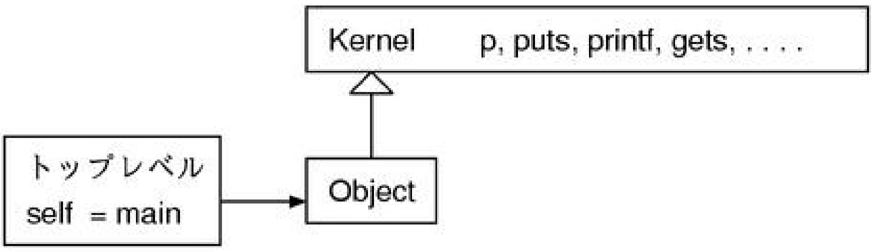
p(lvar)   # 输出"toplevel"
```

表示上下文的self

以前执行方法的时候自己自身(调用方法的对象)会成为self。虽然这个是正确的说法，但是话只说了一半。实际上不管是在Ruby程序执行到哪里，self总会被具体赋值。也就是说在顶层和在类定义语句中都会有self存在。

顶层是存在self的，顶层的self就是main。没有什么奇怪的规则，main就是Object的实例。其实main也仅仅是为了self而存在的，并没有什么深入的含义。

也就是说顶层的self是main，main则是Object的实例，从顶层也可以直接调用Object的方法。另外在Object中引入了Kernel这个模块，里面存在着诸如 `p` 和 `puts` 一样函数风格的方法。于是在顶层也可以调用 `p` 和 `puts` 。



当然 `p` 本来就不是函数而是方法。只是因为其定义在Kernel中，无论在哪里，换句话说无论self的类是什么，都可以被当作"自己的"方法来像函数一样被调用。所以Ruby在真正意义上不存在函数。存在的只有方法。

顺带一提，出了 `p` 和 `puts` 之外，带有函数风格的方法还有 `print puts printf sprintf gets forks exec` 等等等等。大多数都是仿佛曾经在哪里见过的名字。从这个命名中大概也可以想象得出Ruby的性格了吧。

接下来，既然self在那里都会被设定那么在类的定义语句里面也是一样的。在类的定义中self就是类自身(类对象)。于是就会变成这样。

```
class C
  p(self)   # C
end
```

这样设计有什么好处？实际上有个使其好处突出得十分明显的例子。请看。

```
module M
end
class C
  include M
end
```

实际上这个include是针对类对象C的调用。虽然我们还没有提及，但是很明显Ruby可以省略调用方法的括号。由于到目前还没有结束类定义语句的内容，所以作者为了尽量使其看起来不像方法的调用，而把括号给去掉了。

载入

Ruby中载入库的过程也是在执行中进行。通常这样写。

```
require("library_name")
```

为了不被看到的假象所欺骗这里再说一句，require其实是方法，甚至都不是保留语句。这样写的话在写的地方被执行，库里面的代码得以被执行。因为Ruby中不存在像Java里面的包的概念，如果想要将库的命名空间分开来的话，可以建立一个文件夹然后将库放到文件夹里面。

```
require("somelib/file1")
require("somelib/file2")
```

于是在库中也可以定义普通的类和模块。顶层的常量作用域和文件并没有多大关联，而是平坦的，于是从一开始就可以获取在其他文件中定义的类。如果想要用命名空间来区分类的名字，可以用下面的方法来显式嵌入模块。

```
# net 库的例子来使用模块命名空间来区分类
module Net
  class SMTP
    # ...
  end
  class POP
    # ...
  end
  class HTTP
    # ...
  end
end
```

就类的话题继续深入

还是关于常量

到目前为止用文件系统的例子来类比了常量的作用域。现在请你完全忘掉刚才的例子。

常量里面还有各种各样的陷阱。首先，我们可以获取"外层"类的常量。

Const = "ok" class C p(Const) # 输出"ok" end

为什么要这样？因为可以方便使用命名空间来调用模块。到底怎么回事？我们来用之前Net库的类来做进一步说明。

```
module Net
  class SMTP
    # 在方法中可以使用Net::SMTPHelper
  end
  class SMTPHelper   # 辅助Net::SMTP的类
  end
end
```

在这种场合，在SMTP类中只需要写上`Net::SMTPHelper`就可以进行调用。于是就有了"外层类如果可以调用的话就很方便了"这个结论。

不管外层的类进行了多少层嵌套都可以调用。在不同的嵌套层次中如果定义了同名的常量，会调用从内而外最先找到的变量。

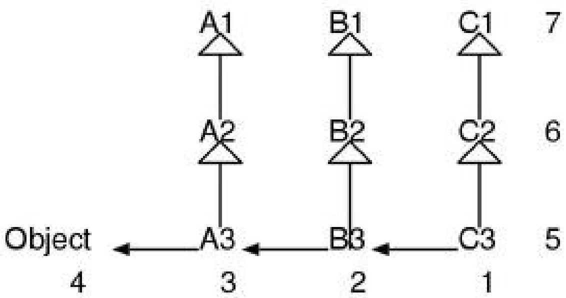
```
Const = "far"
class C
  Const = "near" # 这里的Const比最外层的Const更近一些
  class C2
    class C3
      p(Const)   # 输出"near"
    end
  end
end
```

真是太特么复杂了。

我们来总结一下吧。在探索常量的时候，首先探索的是外层的类，然后探索父类。例如看下面这个故意写成这么扭曲的例子。

```
class A1
end
class A2 < A1
end
class A3 < A2
  class B1
  end
  class B2 < B1
  end
  class B3 < B2
    class C1
    end
    class C2 < C1
    end
    class C3 < C2
      p(Const)
    end
  end
end
```

在C3内部想要获取Const的话，会按照下图的顺序进行探索。



注意一点，外层的类的父类，例如A1和B2是不会被探索的。探索的时候所谓的外层仅仅是向外层，父类的话也仅仅是朝着父类的方向。如果不这样的话，会造成探索的类过多，而无法正确预测这个复杂的行为。

meta类

我们说过对象可以调用方法。调用的方法则是对象的类所决定的。那么类对象也一定存在自己所属的类。



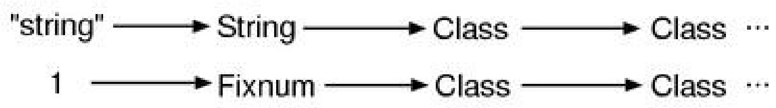
这个时候直接和Ruby确认是最好的方法。返回自己所属的类的方法是`Object#class`。

```
p("string".class())   # 输出String
p(String.class())     # 输出Class
p(Object.class())     # 输出Class
```

String貌似属术语Class这个类的。那么进一步Class所属的类是什么呢？

```
p(Class.class())      # 输出Class
```

又是Class。也就是说不管是什么对象，沿着 `.class().class().class()……` 总会到达Class，然后陷入循环以致最后。



Class是类的类。我们称拥有"xx的xx"的这种递归构造的的东西"meta xx"。所以Class也被叫做"meta class(类)"。

meta对象

接下来我们改变目标，来考察一下模块。模块也是对象，当然会有其所属的类了。我们来调查一下。

```
module M
end
p(M.class())   # 输出Module
```

模块对象的类貌似是Module。那么Module类对象类是什么呢？

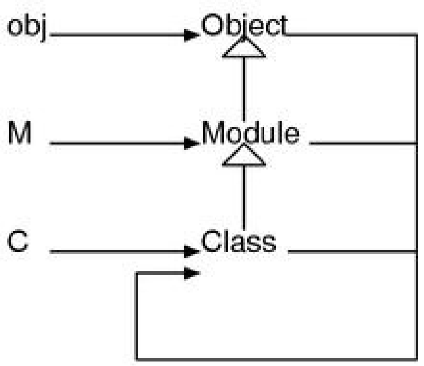
```
p(Module.class())   # Class
```

答案又是Class。

接下来还个方向继续调查其中的继承关系。Class和Module的父类到底是什么？Ruby中可以通过 `Class#superclass` 来进行查看。

```
p(Class.superclass())    # Module
p(Module.superclass())   # Object
p(Object.superclass())   # nil
```

Class居然是Module下层的类。根据以上事实可以得到Ruby中重要类的关系图。



到目前为止我们没有做任何说明就是用了new和include这些东西。现在终于可以解释清楚他们的真实面貌了。new实际上是Class类定义的方法。所以无论是什么类(因为都是Class类的实例)都可以使用new。但是Module里面没有定义new所以无法生成实例。同样，include被定

义在Module类中，所以不管是类还是模块都可以调用include。

奇异方法

对象可以调用方法，能调用的方法由对象所属的类来决定，到目前为止我们都这么说。但是作为设计理念我们还是希望方法都是属于对象的。说到底只是因为同样的类定义同样的方法可以省去不少麻烦。

于是实际上在Ruby中是存在不通过类而直接给对象定义的方法的机制的。可以这样写。

```
obj = Object.new()
def obj.my_first()
  puts("My first singleton method")
end
obj.my_first()   # My first singleton method
```

正如你已了解到，Object是所有类的根目录。这么重要的类里面是不会轻易让你定义my_first这种奇怪名字的方法的。另外obj是Object的实例。但是却可以通过obj这个实例来调用my_first这个方法。也就是说，我们定义了和所属类完全无关的方法。这种给对象定义的方法我们称为奇异方法(singleton method)。

那么什么时候使用奇异方法呢？首先是像Java和C++一样定义静态方法的时候。也就是说不需要生成实例就可以使用的方法。这种方法在Ruby里面作为Class类对象的奇异方法而存在。

例如UNIX中存在unlink这种系统调用来删除文件的别名。Ruby中这个方法作为File类的奇异方法可以直接被使用。我们来试一试。

```
File.unlink("core")  # 消去core名称的dump。
```

每次提到都说一遍"这个是File对象的奇异方法unlink"实在是太麻烦了，下面用"File.unlink"代替。注意不要写成"File#unlink"或者是将"在File类中定义的方法write"错写成"File.write"。

下面是写法的总结

写法	调用的对象	调用的实例
File.unlink	File类自身	File.unlink("core")
File#write	File的实例	f.write("str")

类变量

类变量是Ruby1.6新增的内容。类变量和变量属于同一个类，可以被类以及类的实例代入，引用。来看一下例子。开头是"@@"就是类变量。

```
class C
  @@cvar = "ok"
  p(@@cvar)      # 输出"ok"

  def print_cvar()
    p(@@cvar)
  end
end

C.new().print_cvar()  # 输出"ok"
```

类变量也是由最初的定义来赋值代入的，所以在代入之前引用出错。仿佛多加了个"@"就和一般的实例变量不一样了。

```
% ruby -e '
class C
  @@cvar
end
'
-e:3: uninitialized class variable @@cvar in C (NameError)
```

在这里直接用"-e"命令执行了程序。"'"之间三行是程序正文。

另外类变量是可以继承的。换句话说，子类可以代入，引用父类的类变量。

```
class A
  @@cvar = "ok"
end

class B < A
  p(@@cvar)            # "ok"
  def print_cvar()
    p(@@cvar)
  end
end

B.new().print_cvar()   # "ok"
```

全局变量

最后姑且说一句，全局变量也是存在的。无论在程序的哪里都可以代入，引用。在变量前加上"$"就成了全局变量。

```
$gvar = "global variable"
p($gvar)   # "global variable"
```

全局变量和实例变量一样，我们可以看作所有的名称都是在代入之前就被定义好的。所以就算是代入前我们引用这个变量，只会返回nil而不会发生错误。

(第一章完)

第二章 对象

对象

点睛

本章开始我们终于要开始探索ruby的源码了。首先按照预告我们先从对象的构造开始。

接下来我们来考虑一下对象作为对象而成立的必要条件吧。 之前已经说明过几次对象到底是什么，其实作为对象不可缺少的条件有三个，分别是

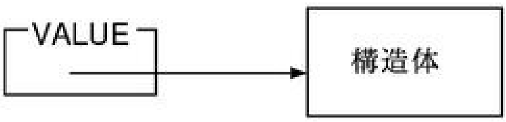
- 可以区别自己和外界。(拥有识别标志)
- 能够对外界的行动作出反应。(方法)
- 拥有内部状态。(实例变量)

本章会对这三个特征顺序确认。

主要注目的文件是ruby.h，其他诸如object.c, class.c, variable.c也会稍微关注。

value和对象的结构体

ruby中对象的实体是由结构体来表现的。各种处理经常要和指针打交道。 结构体根据不同的类会使用不同的类型，但是指针无论是在哪个结构体中 都是VALUE类型。



VALUE的定义如下。

```
  71  typedef unsigned long VALUE;
(ruby.h)
```

VALUE实际上被用来强制转换成各种对象结构体的指针。所以当指针的长度和unsigned long不一致的时候ruby就无法正常工作。严格意义上来说，如果存在比sizeof(unsigned long)更长的指针类型ruby会无法正常工作。当然最近的系统基本上都符合上述要求，过去不符合这种要求的机器还是很多的。

至于结构体，也是有很多种类。主要是按照对象的类来进行区别。

struct RObject 凡是不符合以下条件的

struct RClass Class对象

struct RFloat 小数

struct RString 字符串

struct RArray 数组

struct RRegexp 正则表达式

struct RHash 哈希表

struct RFile IO、File、Socket之类

struct RData 上述以外所有用C语言定义的类

struct RStruct Ruby的结构体Struct类

struct RBignum 大整数

我们来看几个对象结构体的定义的例子。

```
     /* 用作一般对象的结构体 */
 295 struct RObject {
 296     struct RBasic basic;
 297     struct st_table *iv_tbl;
 298 };

     /* 字符串的结构体（String实例） */
 314 struct RString {
 315     struct RBasic basic;
 316     long len;
 317     char *ptr;
 318     union {
 319         long capa;
 320         VALUE shared;
 321     } aux;
 322 };

     /* 数组（Arrayの实例）的结构体 */
 324 struct RArray {
 325     struct RBasic basic;
 326     long len;
 327     union {
 328         long capa;
 329         VALUE shared;
 330     } aux;
 331     VALUE *ptr;
 332 };

(ruby.h)
```

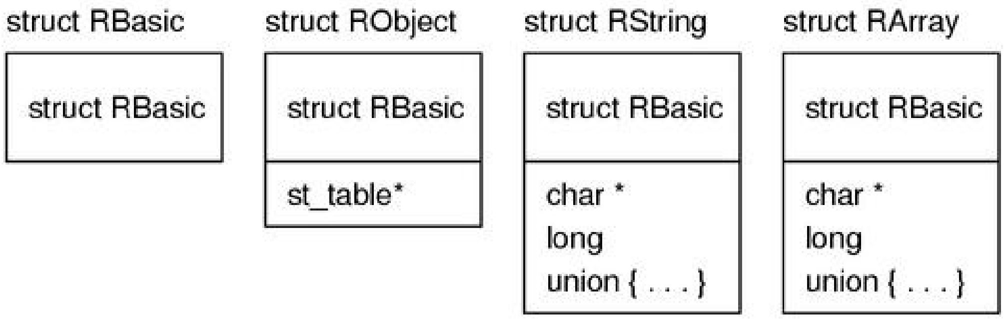
我们先把逐个详细的说明放到后面，首先从整体层面上来说。

首先VALUE被定义成unsigned long类型，要作为指针使用就必须进行强制转换。

于是各个对象的构造函数就配有Rxxxx()这种形式的宏。比如struct RString的宏就是RSTING(),struct RArray的宏是RARRAY()。这些宏的用法如下。

```
VALUE str = ....;
VALUE arr = ....;
RSTRING(str)->len;   /* ((struct RString*)str)->len */
RARRAY(arr)->len;    /* ((struct RArray*)arr)->len */
```

接下来我们看到所有的对象结构体的开头都会有一个struct RBasic类型的成员basic存在。结果就是无论VALUE是指向何种对象的结构体的指针，只要被强制转换成struct RBasic*，就可以访问basic的内容。



既然这么费劲心思作出这种设计，struct RBasic一定存放着Ruby对象的重要信息。我们来看struct RBasic的定义。

```
 290  struct RBasic {
 291      unsigned long flags;
 292      VALUE klass;
 293  };

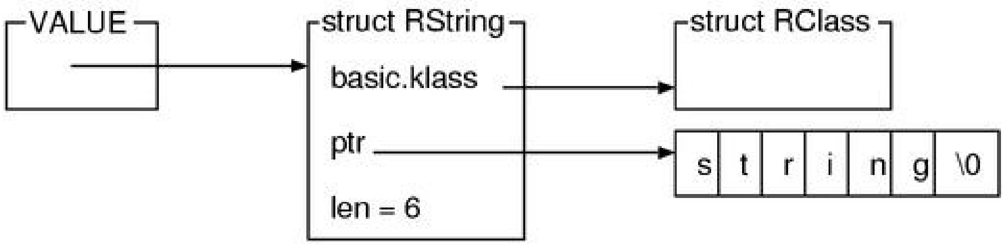
(ruby.h)
```

flags是拥有多种目的的标志，最重要的用途就是保存结构体的类型(struct RObject之类)。表示类型的标志用T_xxxx来定义。可以从VALUE通过宏TYPE()访问。比如下面的例子

```
VALUE str;
str = rb_str_new();    /* 生成ruby的字符串(对应结构体为RSting) */
TYPE(str)              /* 返回值为T_STRING */
```

这些标志都是T_xxxx的形式，struct RString 的话就是T_STRING，struct RArray的话就是T_ARRAY，对应方式十分规则。

struct RBasic的另一个成员klass保存的是对象所属的类。klass的类型是VALUE，足以说明其保存的是Ruby的对象(其实是对象的指针)。也就是说这个就是class类的对象了。



对象和类的关系在本章“方法”这一小节中会详细说明。

顺带一提成员名用klass代替class是为了防止用C++的编译器编译时和保留词class发生冲突。

关于结构体的类型

我们说过struct Basic的成员flags保存了结构体的类型。可是为什么必须保存结构体的类型呢？这是因为所有类型的结构体都是通过VALUE来处理的。当指向结构体的指针被转换成VALUE之后变量中已经不存在类型的信息，编译器也是不会特殊照顾的。于是我们只有自己管理好各自的类型了。这个也是针对所有结构体类型统一管理的一大缺陷。

那么，既然使用的结构体是由类决定的，那么为什么还要把结构体和类分开保存呢？直接从类中访问结构体的类型不就好了吗？不这样做有两个理由。

第一，抱歉我们要推翻刚才说过的话。实际上存在着不具有struct RBasic的结构体(即不存在klass成员)。比如说将在第二部分登场的struct RNode。但是这种特殊的结构体开头还是会有一个flags类型的成员。所以所有结构体只需要拥有flags就可以进行统一管理。

第二，其实类和结构体不是一一对应的。比如说用户用Ruby语言定义的类的实例全部是使用struct RObject来保存。如果要从类中访问结构体的类型就必须记录下所有类和结构体的对应关系。那还不如直接将类的信息放入结构体中来的快捷便利。

basic.flags的用途

谈到basic.flags的用途，刚才一直说是保存结构体类型，这种说法有点恶心我们还是用图来进行一下说明。此图仅仅是为了在之后遇到疑惑的时候可以方便参考，现在还不必全部理解。

！

仅仅是看图的话我们会发现32比特中还有21比特的空余。其实那些部分被定义为FL_USER0~FL_USER8这一系列的标志，根据结构体不同使用目的也不相同。上图为了做示范把FL_USER0也放了进去。

VALUE的填充对象

我们说过VALUE本质上是unsigned long。因为VALUE仅仅是指针，貌似 void*也能胜任，实际上不这样做是有理由的。因为VALUE也有不是指针的时候。非指针的VALUE存在以下六种情况。

- 数值较小的整数
- 符号(symbol)
- true
- false
- nil
- Qundef

我们按顺序来说。

数值较小的整数

Ruby中一切都是对象所以整数也是对象。但是整数的实例是在是太多了，如果每个整数都用一个结构体来表示的话那运行速度就太慢了。假如要递加从0到50000的整数，仅仅如此就要生成50000个对象的话会让人一瞬间陷入犹豫。

那么我们来实际看一下C中奖int类型变换成Fixnum的宏INT2FIX吧，我们来确认一下Fixnum的确是被填埋到了VALUE里面。

```
 123  #define INT2FIX(i) ((VALUE)(((long)(i))<<1 | FIXNUM_FLAG))
 122  #define FIXNUM_FLAG 0x01

(ruby.h)
```

左移一位之后和1相或。

```
110100001000        变换前
1101000010001       变换后
```

就是如此，保证了保存Fixnum的VALUE总是奇数。另一方面，Ruby对象结构体内存空间的申请使用的是malloc()。一般总是会被分配到4的倍数的地址。所以地址的值和保存Fixnum的VALUE值的范围是不会重叠的。

另外，将int和long变换成VALUE还有其他一些宏，比如INT2NUM(),lONG2NUM()。它们都是以“○○2○○”的形式，NUM的话可以同时处理Fixnum和Bignum。比如INT2NUM()会把超过Fixnum范围的数转换成Bignum。NUM2INT()会把Fixnum和Bignum都转换成int类型。如果超出int的范围会发生例外，没有必要在这里进行越界检测。

符号(symbol)

符号是什么？

这个问题比较麻烦，我们先来说为什么需要符号。首先我们知道ruby内部存在着ID类型的变量。

```
   72  typedef unsigned long ID;
(ruby.h)
```

这个ID是和任意的字符串一一对应的整数。虽然话这么说，但是也不可能和这个世界上所有的字符串都一一对应吧？这种对应关系仅仅是存在于"ruby的进程之中"。关于ID的访问方法我们在下一章"名称和命名表"中讲述。

语言处理的程序需要处理大量的名字。变量名，常量名，类名，文件名等等。这些大量的名字如果用char*来保存处理实在是太不容易了。如果你硬要问是哪里不容易，我可以告诉你那就是除了内存管理还是内存管理。另外名字的比较也经常会发生，如果每次都比较字符串是否相符的话实在是效率太低。于是我们不直接处理字符串，而是将其对应到别的东西上面来处理。而那“别的东西”就是整数了。因为整数的处理是最简单的。

将ID带入到ruby的世界中的正是符号。ruby1.4之前直接将ID的值转换成Fixnum作为符号使用。现在也可以通过Symbol#to_i来访问它的值。然而在实际运用中逐渐发现把符号当作Fixnum处理实在不太妥当，于是在ruby1.6之后就有了独立的符号类Symbol了。

符号对象由于经常作为哈希表的键使用所以数量非常多。于是Symbol就和Fixnum一样被填埋进了VALUE里面。我们来看一下将ID转换成Symbol的宏ID2SYM()。

```
 158  #define SYMBOL_FLAG 0x0e
 160  #define ID2SYM(x) ((VALUE)(((long)(x))<<8|SYMBOL_FLAG))

(ruby.h)
```

左移8bit就相当于乘上256，也就是4的倍数。然后和0x0e相或(这个时候和加法没区别)，这样的话最终结果也不会是4的倍数。当然也不会是奇数，也就是说不会和其他的VALUE类型相重叠。真是巧妙的方法。

最后我们来看一下ID2SYM()的逆变化SYM2ID()。

```
 161  #define SYM2ID(x) RSHIFT((long)x,8)
(ruby.h)
```

RSHIFT是右移。右移根据平台不同会出现整数的符号剩余，不剩余的区别，所以保险起见我们用宏来代替。

true false nil

这三个是Ruby里面特殊的对象。分别是代表逻辑真，逻辑假，和没有对象的对象。这三者的c语言中的值定义如下。

```
 164  #define Qfalse 0        /* Rubyのfalse */
 165  #define Qtrue  2        /* Rubyのtrue */
 166  #define Qnil   4        /* Rubyのnil */

(ruby.h)
```

这次竟然是偶数了。但是要注意0和2作为指针使用是不可能的。所以不必担心和其他的VALUE值重复。因为虚拟内存空间第一块地址通常不会被分配。同时也是为了当想要访问NULL指针的时候程序能够迅速得出错。

另外Qfalse因为值为0所以才c语言层次也是被当作逻辑假来使用的。实际上在ruby的返回逻辑真假值的函数中，返回值会被转换成VALUE或者init然后返回Qtrue/Qfalse。这种做法很常见。

至于Qnil，有专门针对VALUE判断其是否为Qnil的宏。NIL_P()

```
 170  #define NIL_P(v) ((VALUE)(v) == Qnil)
(ruby.h)
```

~p这种命名方式术语lisp风格。其表示进行的处理是返回真值的行为。也就是说NIL_P其意思"为参数是否为nil？" p来自于predicate。(断言/谓语)。这种命名规则在ruby中被广泛使用。

另外ruby中除了false和nil是假其他都是真。但是c中的话nil(Qnil)也是真。于是用c语言判定Ruby表达式真假的宏RTEST()应该如下。

```
169  #define RTEST(v) (((VALUE)(v) & ~Qnil) != 0)
(ruby.h)
```

Qnil只有第三位的比特是1,~P取反之后只有第三位是0。与其bit 和之后结果是真的只有Qfalse和Qnil。

加上!=0是为了确保结果是0或者1。因为glib这个库要求真值只能是0或者1。（[ruby-dev:11049]）

说来Qnil的Q是什么玩意儿？是R的话还可以理解，为什么是Q呢？向人询问的结果是因为emacs中是这样的。比预料中有趣呢。

Qundef

```
 167  #define Qundef 6                /* undefined value for placeholder */
(ruby.h)
```

这个值在解释器内部作为“未定义值”来使用。在Ruby中是不会出现的。

方法

Ruby对象最重要的性质要数拥有自我身份，能够调用方法，以及按照实例存有数据这三个方面了。这个小节我们来讲第二点，对象和方法结合的方式。

struct RClass

在ruby中类也是作为对象存在的。那当然类的对象的实体也需要一个结构体。这个结构体就是struct RClass了。这个结构体类型的flag为T_CLASS。

另外类和模块基本属于同一概念所以没有必要区分各自的实体。于是模块的结构体也是用struct RClass来表现的。模块的结构体flag被设置成T_MODULE来进行区别。

```
300  struct RClass {
301      struct RBasic basic;
302      struct st_table *iv_tbl;
303      struct st_table *m_tbl;
304      VALUE super;
305  };

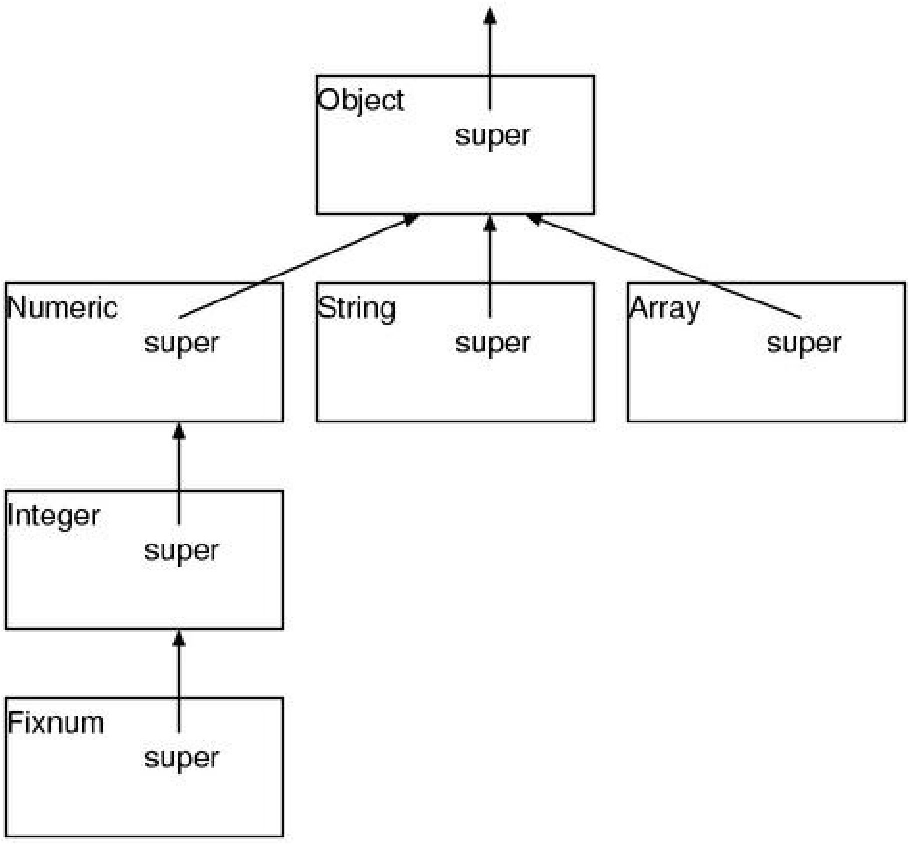
(ruby.h)
```

首先注意到m_tbl(Method Table)这个成员。struct st_table是ruby中到处可见的哈希表。详细会在下一章“名称与命名表”中说明，总之就当作他是记录一对一关系的东西就好了。m_tbl就是用来记录这个类所有的方法的名称(ID)和方法实体直接对应关系的。关于method实体的构造方法我们会在第二部，第三部进行解说。

接下来第四个成员super正如字面意思，保存着父类的信息。因为是VALUE所以指向的是父类的类的对象(指针)。Ruby中不存在父类的类只有Ojbect。(译者注:在ruby1.6之前是如此)

实际上Object里面的所有方法都定义在Kernel这个模块中。Object只是将其引入。这个之前我们也已经讲过了。模块的功能几乎和多重继承相当，一见似乎只是用super的话无法表现一些复杂的关系，ruby中正是用了巧妙的方法使其看起来只是单一继承。这个操作我们会在第四章“类和模块”中说明。

另外受这个变化的影响Object的结构体的super的内容是Kernel实体的struct Rclass， 后者的super被定义成NULL。换句话说，super如果是NULL的话RClass就是Kernel的实体。



方法的搜索

既然类呈现这样的构造方式那么该如何调用方法也可以很容易想象了。首先探索对象类的m_tbl，如果没有找到就继续顺着super在父类中的m_tbl中寻找，依次回溯。也就是说如果知道Object也没有找到该方法，那么该方法就是还未定义。

按照以上的顺序来探索m_tbl的方法请见search_method()。

```
 256  static NODE*
 257  search_method(klass, id, origin)
 258      VALUE klass, *origin;
 259      ID id;
 260  {
 261      NODE *body;
 262
 263      if (!klass) return 0;
 264      while (!st_lookup(RCLASS(klass)->m_tbl, id, &body)) {
 265          klass = RCLASS(klass)->super;
 266          if (!klass) return 0;
 267      }
 268
 269      if (origin) *origin = klass;
 270      return body;
 271  }

(eval.c)
```

这个函数在类对象klass中寻找名称为id的方法。

`RCLASS(value)` 的内容为 `((struct RClass*)(value))` 的宏。

st_lookup()是用来在st_table中检索和键对应的值的函数。如果找到值就返回真，并将找到的值写入第三个地址参数(body)。这边这些函数在卷尾的函数参照中会有记载，如有需要可以随时参考。

话说来每次都进行探索的话实在是太慢了。实际上被调用的参数会被保存到缓存中，第二次就不必每次都用super方法来回溯寻找了。包括缓存的搜索我们会在第十五章“方法”中进行介绍。

实例变量

这章节我们介绍成为对象必须条件的第三点，实例变量的实装。

rb_ivar_set()

实例变量是一种以对象为单位存放其特有的数据的方式。既然是对象特有那么好像将数据保存到对象内部(对象的结构体)会比较好？那么实际上是个什么情况呢？我们来看一下将实例变量带入对象中的函数rb_ivar_set()一探究竟。

```
     /* 向obj对象的成员变量id中代入val */
 984  VALUE
 985  rb_ivar_set(obj, id, val)
 986      VALUE obj;
 987      ID id;
 988      VALUE val;
 989  {
 990      if (!OBJ_TAINTED(obj) && rb_safe_level() >= 4)
 991          rb_raise(rb_eSecurityError,
                       "Insecure: can't modify instance variable");
 992      if (OBJ_FROZEN(obj)) rb_error_frozen("object");
 993      switch (TYPE(obj)) {
 994        case T_OBJECT:
 995        case T_CLASS:
 996        case T_MODULE:
 997          if (!ROBJECT(obj)->iv_tbl)
                  ROBJECT(obj)->iv_tbl = st_init_numtable();
 998          st_insert(ROBJECT(obj)->iv_tbl, id, val);
 999          break;
1000        default:
1001          generic_ivar_set(obj, id, val);
1002          break;
1003      }
1004      return val;
1005  }

(variable.c)
```

rb_raise()和rb_error_frozen()都是错误检测。这之后我们会反复强调，错误检测虽然在现实中是需要的，但是不是处理的本质。所以第一次读代码的时候可以将错误处理完全忽略。

去掉了错误处理之后就只剩下了swtich语句。类似

```
switch (TYPE(obj)) {
  case T_aaaa:
  case T_bbbb:
     :
}
```

的形式是ruby特有的书写习惯。TYPE()返回对象构造体的类型(T_OBJECT和T_STRING之类)的宏。类型flag因为是整数所以完全可以使用switch分支。Fixnum和Symbol虽然不存在构造体，但是会在TYPE()中j进行特殊处理进而返回T_FIXNUM和T_SYMBOL，所以大可不必担心。

接下来我们回到`rb_ivar_set()`。好像就只有T_OBJECT T_CLASS T_MODULE这三个类型的处理是分开来单独进行的。这三个被选中是因为他们的结构体的第二个成员是iv_tbl。我们来实际确认一下。

```
    /* TYPE(val) == T_OBJECT */
 295  struct RObject {
 296      struct RBasic basic;
 297      struct st_table *iv_tbl;
 298  };

      /* TYPE(val) == T_CLASS or T_MODULE */
 300  struct RClass {
 301      struct RBasic basic;
 302      struct st_table *iv_tbl;
 303      struct st_table *m_tbl;
 304      VALUE super;
 305  };

(ruby.h)
```

iv_tbl对应的是Instance Variable Table，也就是实例变量表。里面记录的是实例变量名和对应的值。

我们再来贴一下rb_ivar_set()中，当结构体拥有iv_tbl成员时的处理代码。

```
if (!ROBJECT(obj)->iv_tbl)
    ROBJECT(obj)->iv_tbl = st_init_numtable();
st_insert(ROBJECT(obj)->iv_tbl, id, val);
break;
```

ROBJECT()是用来将VALUE强制转换成struct RObject*的宏。obj指向的也有可能是struct Rclass，但是仅仅是访问第二成员变量的话是不会发生什么问题的。

st_init_numtable()是新生成st_table的函数。st_insert()是在st_table中生成关联的函数。

综上所述这段代码所做的事情，就是当iv_table()不存在的时候新建一个，然后将对应的"变量名=>对象"记录到其中。

注意一点，struct Rclass自身是类对象的结构体，所以其变量表也是类对象自己的东西。用ruby程序来说，就如下面这种情况。

```
class C
  @ivar = "content"
end
```

generic_ivar_set()

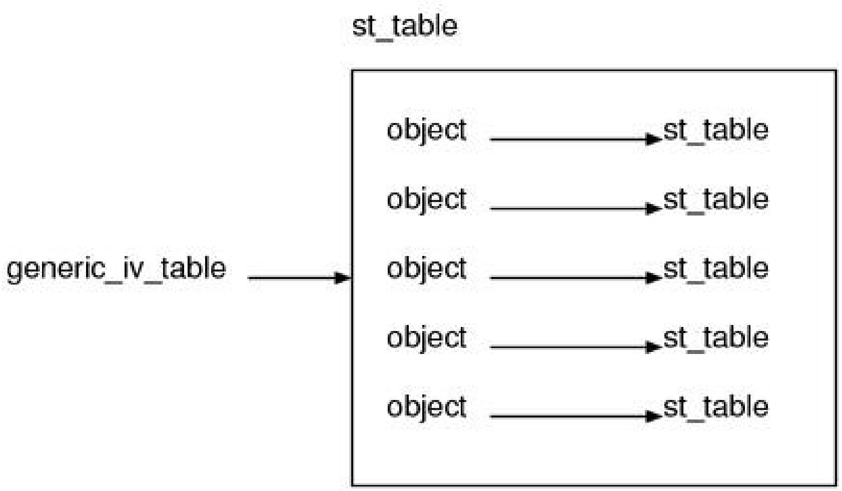
向T_OBJECT T_MODULE T_CLASS之外的结构体代入变量会是怎么样呢？

```
#没有iv_tbl的结构体
1000  default:
1001    generic_ivar_set(obj, id, val);
1002    break;

(variable.c)
```

这个时候处理会交给generic_ivar_set()。看具体的函数之前先把大框架说明一下吧。

T_OBJECT T_MODULE T_CLASS以外的构造体是没有iv_tbl成员的(之后会说明为什么没有的理由)。但是就算是没有该成员也可以通过别的手段将实例和struct st_table对应起来。ruby将这种对应关系保存在全局的st_table，即generic_iv_table中来解决此问题。



我们来看一下实际的代码。

```
 801  static st_table *generic_iv_tbl;

 830  static void
 831  generic_ivar_set(obj, id, val)
 832      VALUE obj;
 833      ID id;
 834      VALUE val;
 835  {
 836      st_table *tbl;
 837
          /* 总之可以先无视 */
 838      if (rb_special_const_p(obj)) {
 839          special_generic_ivar = 1;
 840      }
          /* 不存在generic_iv_tbl则新建 */
 841      if (!generic_iv_tbl) {
 842          generic_iv_tbl = st_init_numtable();
 843      }
 844
          /* 核心处理 */
 845      if (!st_lookup(generic_iv_tbl, obj, &tbl)) {
 846          FL_SET(obj, FL_EXIVAR);
 847          tbl = st_init_numtable();
 848          st_add_direct(generic_iv_tbl, obj, tbl);
 849          st_add_direct(tbl, id, val);
 850          return;
 851      }
 852      st_insert(tbl, id, val);
 853  }

(variable.c)
```

rb_special_const_p()当参数不是指针的时候返回真。这个if语句里面的内容如果不理解垃圾回收机制的话是无法说明的，所以我们在这里还是先省略。 请读者在阅读第五章的垃圾回收机制之后再自行理解。

st_init_numtable()刚才也出现了，是用来新建一个哈希表。

st_lookup()用来查找和key对应的值。这个时候会查找和obj所关联的实例变量表。如果找到对应的值函数整体就返回眞，在第三个地址参数(&tbl)中记录找到的对应值。也就是说!st_lookup(...)将的是当没有找到对应记录之后发生的事情。

st_insert()刚才也说过了。用来将新的对应记录保存到表中。

st_add_direct()和st_insert()几乎相同，区别在于后者会在追加保存记录之前先检查一下键是否已经存在。也就是说如果使用st_add_direct()来添加新记录的话，如果已经有相同的键存在于表中，会出现一个键对应两个记录的情况。

所以能直接使用st_add_direct()的情况，基本上是刚刚确认过键不存在，或者是刚刚建立新的表这两种情况。这段代码是符合这些情况的。

FL_SET(obj, FL_EXIVAR)是用来设置obj的basic.flags为FL_EXIVAR的宏。basic.flags的所有flag都是FL_xxx的形式，可以通过FL_SET来设置flag。相反用来取消对应flag的宏叫做FL_UNSET()。另外，通常认为FL_EXIVAR的EXIVAR是external instance variable的简称。

插入这个flag是为了提高访问实例变量的速度。如过发现没有这只FL_EXIVAR这个falg，则不用通过探索generic_iv_tbl也可以知道实例变量不存在。相比之下当然是bit的校验比起探索struct st_table来的速度更快。

结构体的间隙

现在我们了解了实例变量的保存方式，那么为什么存在没有iv_table的结构体呢？比如struct RString和struct RArray就没有iv_tbl，这个是为什么？那么干脆直接把实例变量当作RBasic的成员算了。

就结论来说，可以这样做但是不应该如此。实际上这个和ruby的对象管理机制紧密相关。

ruby中字符串的数据(char[])所占用的内存使用malloc来访问。但是对象的结构体是个例外。ruby会统一管理分配，从而访问内存。这个时候如果结构体的种类(大小)参差不齐的话管理起来就十分麻烦。所以将所有的结构体用共用体Rvalue来申明并统一分配管理。共用体的大小会和成员中最大的保持一致，所以如果单独有一个结构体的体积非常大就会造成很大的浪费，于是我们还是希望素有的结构体的大小尽可能保持接近。

最常用的结构体要数struct RString(字符串)了吧。接下来是RArray(数组)，RHash(哈希)，RObject(所有用户定义的类) 。那么问题就来了。struct RObject的内容只有sruct RBASIC和一个指针。而struct RString RHash RArray却已经使用了struct Basic和三个指针的空间。也就是说struct Robject越多，就会造成越多的两个指针空间的浪费。更有甚者，RString如果使用了四个指针，RObject实际上就只占用了共用体的一半不到的空间，实在是太浪费了。

另一方面配置iv_tbl的好处主要是访问速度的提升和内存空间的节省，并且我们也不知道这个功能会不会被频繁使用。实际上在ruby1.2之前根本就没有导入过generic_iv_tbl，所以像String和Array中也不能使用实例变量，就算如此也没发生什么大问题。如果仅仅因为这点好处就浪费大量的空间实在是太蠢了。

所以从以上可以做出结论，因为iv_tbl而增加构造体的体积实在是无奈之举。

```
 960  VALUE
 961  rb_ivar_get(obj, id)
 962      VALUE obj;
 963      ID id;
 964  {
 965      VALUE val;
 966
 967      switch (TYPE(obj)) {
      /*（A）*/
 968        case T_OBJECT:
 969        case T_CLASS:
 970        case T_MODULE:
 971          if (ROBJECT(obj)->iv_tbl &&
                  st_lookup(ROBJECT(obj)->iv_tbl, id, &val))
 972              return val;
 973          break;
      /*（B）*/
 974        default:
 975          if (FL_TEST(obj, FL_EXIVAR) || rb_special_const_p(obj))
 976              return generic_ivar_get(obj, id);
 977          break;
 978      }
      /*（C）*/
 979      rb_warning("instance variable %s not initialized", rb_id2name(id));
 980
 981      return Qnil;
 982  }

(variable.c)
```

结构基本相同。

(A)struct RObject如果是Rclass，则检索iv_tbl。刚才说过了，有可能iv_tbl是NULL，所以不先检查一下的话会出错。接着st_lookup会在找到相应记录的时候返回眞。这个if语句整体来说就是"如果已经代入过此实例变量则返回其值"。

(C)如果没有找到对应的值……也就是说如果想要访问的是还没有被代入的实例变量，那么跳过if和switch，直接运行下面的语句。这个时候会发生rb_warning(),并返回nil。这是因为ruby的实例变量不需要代入也可以被访问。

(B)另外，当结构体既不是struct RObject也不是RClass的时候，首先从generic_iv_tbl中寻找对象的实例变量表。generic_ivar_get()实现的功能不用我说也能想到。另外需要注意的是if语句。

刚才说过将进行过generic_ivar_set()处理的对象插入FL_EXIVAR。在这里这个flag使得运行高速化的特征就显现出来了。

rb_special_const_p()是什么玩意儿？这个函数当obj不存在结构体的时候为眞。因为不存在构造体所以也不需要basic.flags(因为根本无从插入flag)。所以FL_xxx()遇到这种对象总是会返回假。于是这里对待rb_special_const_p()为眞的对象必须格外小心翼翼。

对象的结构体

这小节中我们简要介绍对象结构体最重要的具体内容的的处理方法。

struct RString

```
 314  struct RString {
 315      struct RBasic basic;
 316      long len;
 317      char *ptr;
 318      union {
 319          long capa;
 320          VALUE shared;
 321      } aux;
 322  };

(ruby.h)
```

ptr是指向字符串的指针，len是字符串的长度，非常直观。

Ruby的字符串与其说是字符串不如说是字符序列。因为可以包含NUL在内的任何字符。所以在ruby中就算在终端设置NUL也没有多大意义，不过因为C的函数中要求NUL,所以为了便利还是将NUL设置成字符串的终端。但是NUL是不包含在len之中的。

另外解释器和扩展库中处理字符串的时候可以通过`RSRTING(str)->ptr``RSTRING(str)`来访问ptr和len。但是要注意以下几点。

- 检查str是否确实指向struct RString。
- 可以访问成员但是不能改变其内容
- 不可以将RSTRING(str)->ptr保存到诸如临时变量之中供之后使用。

这是为什么？其中一个原因是软件工程学上的原则。不可以随便修改别人的数据。既然有接口函数就乖乖使用接口函数。不过还有其他不允许擅自访问和保存指针的理由，这和第四个成员变量aux有关。但是要详细说明aux的使用方法又必须得详细说明ruby字符串的一些特征。

ruby的字符串自身是可以改变的(mutable),所谓的变化是指

```
s = "str"          # 生成字符串代入s
s.concat("ing")  # 向字符串s中追加"ing"。
p(s)               # 输出"string"
```

这样s指向的对象的内容就变成了"string"。java和Python的字符串是没有这种特性的。硬要说的话，这个特性和Java的StringBuffer很接近。

接下来我们来看看他们之间到底有什么关系。首先既然可以发生改变自然字符串的长度len也会改变。既然长度发生改变那么这个时候内存的分配就会发生增减。当然也可以使用realloc()，但是malloc和realloc这些操作都太重了，仅仅是为了变更字符串就realloc()一次的话实在是负担太大了。

于是ptr所指向的内存空间通常会比len稍微长一点。这样的话追加的字符串正好能放入多余的内存中就可以不必调用realloc()了，这样的话速度就上去了。结构体中的aux.capa保存的就是这个多余的长度。

那么另一个aux.shared是什么玩意儿呢。这个也是为了提高从字符串序列中生成对象的速度而采用的机制。请看下面的ruby程序。

```
while true do  # 永远地重复
  a = "str"        # 将内容是"str"的字符串放入a
  a.concat("ing")  # 向a中追加"ing"。
  p(a)             # 输出string
end
```

不管是循环多少次总会在第四行的p出输出"string"。放在一般情况下那就得每次通过"str"这个式子新建一个char[]类型的字符串对象。 但是对于字符串通常情况下都不会做任何改变，这个时候就会造成多次复制char[]的资源浪费。于是我们就期望能够共用一个char[]。

作为共用所存在的就是aux.shared这个东西了。使用表达式生成的字符串对象都会共用同一个char[]。只有当真正发生变化时候才会专门去申请内存分配。使用共用的char[]的结构体中的basic.flags标志中会被设立ELTS_SHARED这个标志。aux.shared会保存原来的对象。ELTS是elements的简称。

我们回到RSTRING(str)->ptr的话题中。之所以可以访问但不能改变指针对象是因为这会使得len和capa的值和真实情况不符。另外如果要改变用序列表达式所新建的字符串对象的内容，则需要把对象中的aux.shared成员移除。

最后我们来列举几个使用RString的例子。str可以看成是指向RString的value。

```
RSTRING(str)->len;               /* 长度 */
RSTRING(str)->ptr[0];            /* 第一个字符 */
str = rb_str_new("content", 7);  /* 生成内容是"content"的字符串。
                                    第二个参数是其长度 */
str = rb_str_new2("content");    /* 生成内容是"content"的字符串。
                                    长度会使用strlen()来计算 */
rb_str_cat2(str, "end");         /* 在Ruby字符串中后接C的字符串 */
```

struct RArray

struct RArray是存放Ruby的数组实例的结构体。

```
 324  struct RArray {
 325      struct RBasic basic;
 326      long len;
 327      union {
 328          long capa;
 329          VALUE shared;
 330      } aux;
 331      VALUE *ptr;
 332  };

(ruby.h)
```

除了ptr之外几乎和struct RString一样。ptr指向的是数组的内容，len是其长度。aux的用法和struct RString中介绍的相同。aux.capa是ptr所指向的内存的真正的长度，aux.shared则是当数组为共用的时候指向共用数组的指针。

访问成员的方法也和RString类似。 通过RARRAY(arr)->ptr和RARRAY(arr)->len可以访问成员但是不能改变成员的内容。我们来看一下简单的例子。

```
/* 用c语言操作数组 */
VALUE ary;
ary = rb_ary_new();             /* 生成空的数组 */
rb_ary_push(ary, INT2FIX(9));   /* 将Ruby的9加入数组 */
RARRAY(ary)->ptr[0];            /* 访问编号是0的元素 */
rb_p(RARRAY(ary)->ptr[0]);      /* 输出ary[0]（输出9） */

# 用ruby进行操作
ary = []      # 生成空的数组
ary.push(9)   # 将Ruby的9加入数组
ary[0]        # 访问编号是0的元素
p(ary[0])     # 输出ary[0]（输出9）
```

struct RRegexp

RRepexp是存放正则表达式的结构体。

```
 334  struct RRegexp {
 335      struct RBasic basic;
 336      struct re_pattern_buffer *ptr;
 337      long len;
 338      char *str;
 339  };

(ruby.h)
```

ptr是已经编译好的正则表达式。str是编译前的正则表达式(正则表达式的源码)，len是其长度。

处理Rexgexp对象的代码本书中将不会出现所以这里就省略了。就算要在扩展库中使用，只要不涉及很特殊的用法，参考一些接口函数就应该足够了吧。

struct RHash

struct RHash是哈希表Hash实例所在的结构体。

```
 341  struct RHash {
 342      struct RBasic basic;
 343      struct st_table *tbl;
 344      int iter_lev;
 345      VALUE ifnone;
 346  };

(ruby.h)
```

其实这个是struct st_table的wrapper。关于st_table我们会在下一章"名称和命名表"中详细解说。

ifnone存放的是搜索失败时候使用的键，默认是nil。iter_lev是为了哈希表的re-entrance(多进程安全)存在的。

struct RFile

struct RFile是服务嵌入类Io和其后继子类实例的构造体。

```
 348  struct RFile {
 349      struct RBasic basic;
 350      struct OpenFile *fptr;
 351  };

(ruby.h)
```

```
 19  typedef struct OpenFile {
  20      FILE *f;                    /* stdio ptr for read/write */
  21      FILE *f2;                   /* additional ptr for rw pipes */
  22      int mode;                   /* mode flags */
  23      int pid;                    /* child's pid (for pipes) */
  24      int lineno;                 /* number of lines read */
  25      char *path;                 /* pathname for file */
  26      void (*finalize) _((struct OpenFile*)); /* finalize proc */
  27  } OpenFile;

(rubyio.h)
```

成员几乎都保存在了struct OpenFile中。IO对象的实例并不多所以可以这样存放。各个成员的用途都有些。基本上是C语言stdio的warpper。

struct RData

struct RData和目前为止介绍的东西目的都不一样。这个主要是为了存放扩展类的结构体。

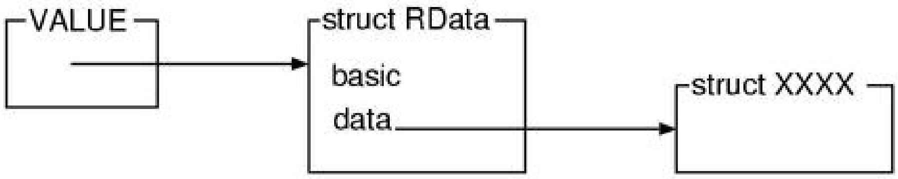
编写扩展库的类的实体当然也需要一个结构体来存放。但是结构体的类型是生成的类所决定的，所以无法事先知道类的大小和结构。于是ruby提供了一个"管理用户自定义的结构体指针的结构体"，这个东西就是struct RData了。

```
 353  struct RData {
 354      struct RBasic basic;
 355      void (*dmark) _((void*));
 356      void (*dfree) _((void*));
 357      void *data;
 358  };

(ruby.h)
```

data是指向用户定义的构造体的指针。dfree是释放自定义构造体的函数，dmark是mark and sweep中进行mark的函数(涉及垃圾回收)

关于struct RDdata的说明现在还不是时候，总之先看图吧。详细的内容我们会在第五章的"垃圾回收"中介绍。



(第二章完)

第三章 名称和命名表

st_table

作为存储方法的表和实例变量的表，st_table已经出现过几次了。本章首先就st_table做详细的说明。

概要

我们已经说过st_table是哈希表。哈希表是保存一对一对应关系的数据结构。这种一对一关系可以是变量名和变量的值，也可以是函数名和函数的实体，等等。

当然除了哈希表也可以用其它的数据结构来表示一一对应关系。比如可以下面这种list的结构体。

```
struct entry {
    ID key;
    VALUE val;
    struct entry *next;  /* 指向下一个元素 */
};
```

但是这种方法很慢。如果存在1000个元素的话，最糟糕的情况下要遍历1000次这个链表。也就是探索的时间和元素的个数是成正比的。这样的话就会很糟糕。所以从很早就考虑了各种解决方法。哈希表就是这个解决方法的一种。也就是说哈希表并不是仅有的方法，但是能够带来高速化的处理。

接下来我们实际来看st_table。注意看，这个库并不是松本先生的原创。

```
   1  /* This is a public domain general purpose hash table package
          written by Peter Moore @ UCB. */

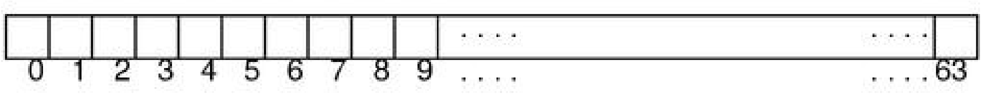
(st.c)
```

……恩至少注释是这么说的。

顺带一提，用谷歌检索到的其他版本的注释上说，st_table是string table的简称。但是我认为general purpose和string到底是有些矛盾的……。

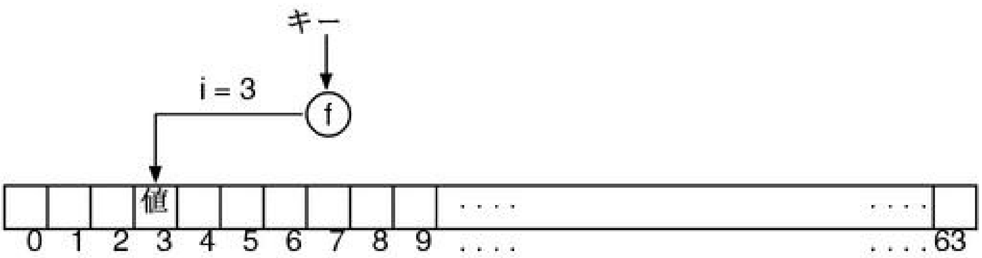
所谓哈希表

哈希表的设想如下。首先有一个长度为n的数组。比如说n=64。



然后准备一个能够将键映射到0到n-1(0~63)的整数i的函数f。这个f被叫做哈希函数。对于同一个键，f必须保证每次都返回相同的i。假设键的值是整数的话，这个整数被64整除的余数肯定是在0到63之间的。所以这个取余的计算可以成为f函数。

要找到对应关系的存储位置的时候，先对键调用哈希函数f，求得i的值，然后数组的第i个元素就好了。也就是说，因为访问数组的某个元素是十分快速的，所以只需要找到某种方法把键转换成整数就好了，这个就是hash的核心思想。

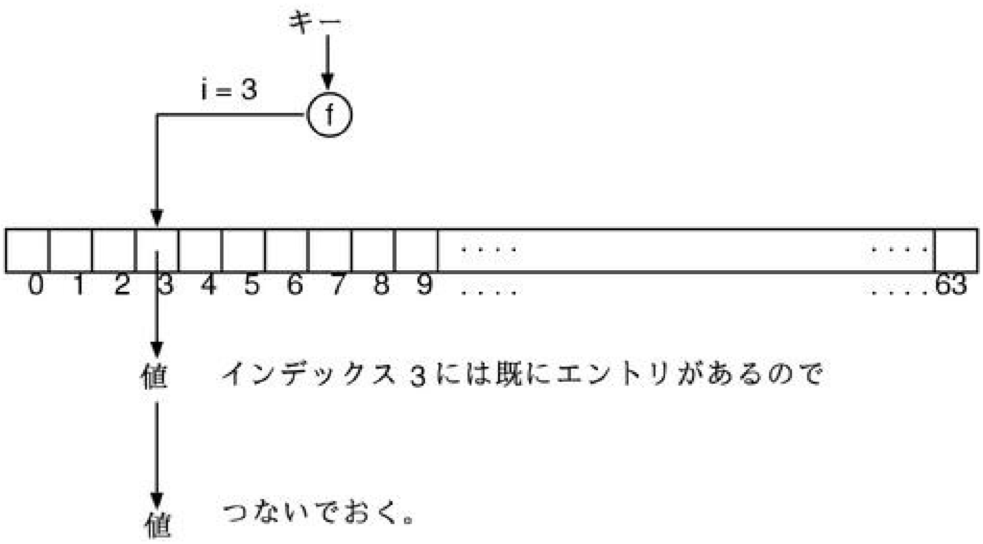


可惜世界上是没有这么理想的情况的。这个方法有个致命的问题。n现在只有64个元素，所以当需要对应64个以上的元素键的话肯定会发生重复。就算键没超过64个，也有可能发生两个键对应相同的索引的情况。比如刚才用64取余的的方法，当键的值是65或者129的时候，对应的哈希值都是1。我们把这个叫做哈希冲突。解决冲突的方法主要有几种。

比如发生冲突的话可以按照下面的方法放入元素。这个方法叫做开放定址法。



另外有一种方法，不是直接将元素存在数组中，而是在数组中存放一个个链表的指针。发生冲突的时候逐渐延长链表的长度。这个方法叫做连锁法。st_table采用的就是这个连锁法。



说来如果能够实现知道键的集合的内容的话也许可以设计出一个绝对不会发生冲突的哈希函数。这个函数被叫做绝对哈希函数。然后还有针对任何的字符串的集合来生成哈希函数的工具，比如说GNU的gperf。ruby的语法解析器实际上也用到了这个工具......还不是说这个的时候。我们会在第二部分继续介绍。

数据结构

现在我们来看实际的代码。在序章中已经说过，如果同时存在数据类型和代码的话当然是先读数据类型。下面我们来看一下st_table实际使用的数据类型。

```
   9  typedef struct st_table st_table;

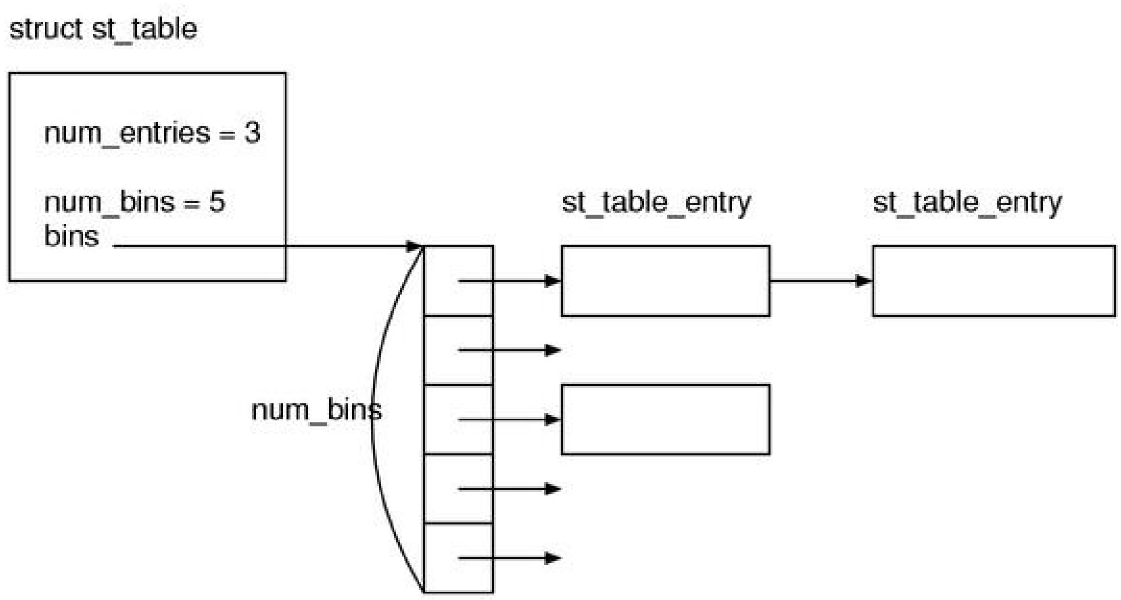
  16  struct st_table {
  17      struct st_hash_type *type;
  18      int num_bins;                   /* 槽的个数 */
  19      int num_entries;                /* 总共存放的元素数*/
  20      struct st_table_entry **bins;   /* 槽 */
  21  };

(st.h)
```

```
  16  struct st_table_entry {
  17      unsigned int hash;
  18      char *key;
  19      char *record;
  20      st_table_entry *next;
  21  };

(st.c)
```

st_table是主体的表的结构，st_table_entry是存放元素的地方。st_table_entry有一个成员叫做next，因为是链表所以当然需要啦。这个就是连锁法的连锁的地方。我们发现其使用了st_hash_type的类型，我们会稍后对此说明首先就别的地方对照下图进行逐一确认好了。



接下来我们来看st_hash_type。

```
  11  struct st_hash_type {
  12      int (*compare)();   /* 比较函数 */
  13      int (*hash)();      /* 哈希函数 */
  14  };

(st.h)
```

毕竟才第三章我们还是认眞得看下去吧。

```
int (*compare)()
```

这个表示的是返回int的函数指针成员compare。hash也是同理。这种变量用下面的方法代入。

```
int
great_function(int n)
{
    /* ToDo */
    return n;
}

{
    int (*f)();
    f = great_function;
```

然后用下面的方法调用。

```
    (*f)(7);
}
```

现在回到st_hash_type的解说。hash compare这个两个成员中，hash就是之前说过的哈希函数。

compare则是用来判断键是否是同一个的函数。因为连锁法中相同的哈希值n的地方会存放多个要素。为了知道其中哪个元素才是我们眞正需要的，这就需要有一个可以完全信任的比较函数。这个比较函数就是compare。

这个st_hash_type是一种很巧妙的通用化方法。哈希表是无法确定自己保存的键的类型的。比如ruby的st_table的键勀是ID，char*，VALUE。如果每种类型都要设计一种哈希的话实在是太蠢了。因为键的类型不同而导致改变的仅仅是哈希函数的部分而已，无论是分配内存还是检测冲突的大部分代码都是相同的。于是我们仅仅把不同的地方作为函数特化起来，用函数指针来制定其具体函数来使用。这样的话就可以使本来就占据着大部分代码的哈希表的实装更加灵活。

面向对象的语言本身就把对象和过程捆绑在一起所以这种构造是没有必要的。或者说这种构造作为语言的功能已经被嵌入进去了。

st_hash_type的例子

st_hash_type的结构体虽然是一种很成功的通用方法，但是也是的代码变得复杂难懂起来。不具体看一下hash和compare函数的话总是没有什么实感。于是现在就可以来看看上一章也出现的st_init_numtable()函数了。这个对应整数键值的哈希函数。

```
182  st_table*
183  st_init_numtable()
184  {
185      return st_init_table(&type_numhash);
186  }

(st.c)
```

st_init_table()是给表分配内存的函数， type_numhash的类型是st_hash_type。type_numhash的内容如下

```
 37  static struct st_hash_type type_numhash = {
 38      numcmp,
 39      numhash,
 40  };

552  static int
553  numcmp(x, y)
554      long x, y;
555  {
556      return x != y;
557  }

559  static int
560  numhash(n)
561      long n;
562  {
563      return n;
564  }

(st.c)
```

实在是太简单。ruby的解释器用的表基本上使用的是type_numhash。

st_lookup()

接下来我们来看哈希结构体里面的函数，从开始可以先从探索函数入手。哈希表中的探索函数st_lookup()内容如下。

```
247  int
248  st_lookup(table, key, value)
249      st_table *table;
250      register char *key;
251      char **value;
252  {
253      unsigned int hash_val, bin_pos;
254      register st_table_entry *ptr;
255
256      hash_val = do_hash(key, table);
257      FIND_ENTRY(table, ptr, hash_val, bin_pos);
258
259      if (ptr == 0) {
260          return 0;
261      }
262      else {
263          if (value != 0)  *value = ptr->record;
264          return 1;
265      }
266  }

(st.c)
```

重要的部分几乎都在do_hash()和FIND_ENTRY()里面，我们按着顺序来看。

do_hash()

```
 68  #define do_hash(key,table) (unsigned int)(*(table)->type->hash)((key))

(st.c)
```

保险起见我们还是把宏里面的复杂的部分单独抽取出来

```
(table)->type->hash
```

这个函数指针带上一个参数key之后就调用了相关的函数。*的内容不是table(而是表示这个带参数的函数指针)。也就是说，这个宏使用按照类型定义的不同的哈希函数type->hash，带上参数key来求哈希值。

接下去我们来看FIND_ENTRY()。

```
 235  #define FIND_ENTRY(table, ptr, hash_val, bin_pos) do {\
 236      bin_pos = hash_val%(table)->num_bins;\
 237      ptr = (table)->bins[bin_pos];\
 238      if (PTR_NOT_EQUAL(table, ptr, hash_val, key)) {\
 239          COLLISION;\
 240          while (PTR_NOT_EQUAL(table, ptr->next, hash_val, key)) {\
 241              ptr = ptr->next;\
 242          }\
 243          ptr = ptr->next;\
 244      }\
 245  } while (0)

 227  #define PTR_NOT_EQUAL(table, ptr, hash_val, key) ((ptr) != 0 && \
          (ptr->hash != (hash_val) || !EQUAL((table), (key), (ptr)->key)))

  66  #define EQUAL(table,x,y) \
          ((x)==(y) || (*table->type->compare)((x),(y)) == 0)

(st.c)
```

COLLISION是用来debug的宏，直接无视好了。

FIND_ENTRY()的参数从前向后分别是

1.st_table

2.应该使用的临时变量

3.哈希值

4.检索的键

第二个参数用来保存查询到的st_table_entry*的值。

另外最外面一层为了保证由多个式子组成的宏安全执行，使用了do～while(0)。这个是ruby，严格来说是c语言的预处理的风格。使用if(1)的话会不小心带上else。而使用while(1)的话最后还需要break。

while(0)的后面不接分号也是有讲究的。如果你硬要问为什么，请看

```
FIND_ENTRY();
```

一般都是这样写，所以最后的就不会多出来一个分号。

st_add_direct()

接下去我们来看在哈希表中添加新数据的函数st_add_direct()。这个函数不检查键是否已经登录，而是无条件得追加新的项目。这个就是direct的意思。

```
308  void
309  st_add_direct(table, key, value)
310      st_table *table;
311      char *key;
312      char *value;
313  {
314      unsigned int hash_val, bin_pos;
315
316      hash_val = do_hash(key, table);
317      bin_pos = hash_val % table->num_bins;
318      ADD_DIRECT(table, key, value, hash_val, bin_pos);
319  }

(st.c)
```

和刚才一样为了求哈希值使用了宏do_hash(),接下来的计算也在FIND_ENTRY()的开头出现过，哈希值就实际的索引号。

然后插入过程自身是依靠ADD_DIRECT执行。从名字就可以看出这是一个宏。

```
268  #define ADD_DIRECT(table, key, value, hash_val, bin_pos) \
269  do {                                                     \
270      st_table_entry *entry;                               \
271      if (table->num_entries / (table->num_bins)           \
                             > ST_DEFAULT_MAX_DENSITY) {      \
272          rehash(table);                                   \
273          bin_pos = hash_val % table->num_bins;            \
274      }                                                    \
275                                                           \
         /*（A）*/                                           \
276      entry = alloc(st_table_entry);                       \
277                                                           \
278      entry->hash = hash_val;                              \
279      entry->key = key;                                    \
280      entry->record = value;                               \
         /*（B）*/                                           \
281      entry->next = table->bins[bin_pos];                  \
282      table->bins[bin_pos] = entry;                        \
283      table->num_entries++;                                \
284  } while (0)

(st.c)
```

开头的if是例外处理的内容，我们稍后看，首先看下面的。

(A)分配st_table_entry，进行初始化

(B)向链表开头追加entry。这个是处理链表的风格。也就是说通过

```
entry->next = list_beg;
list_beg = entry;
```

可以向链表的开头追加元素。这也是Lisp的术语"cons"的意思。就算list_beg是NULL，这段代码也是可以通用的。

最后看一下被我们搁置的代码。

ADD_DIRECT()－rehash

```
 271      if (table->num_entries / (table->num_bins)           \
                             > ST_DEFAULT_MAX_DENSITY) {       \
 272          rehash(table);                                   \
 273          bin_pos = hash_val % table->num_bins;            \
 274      }                                                    \

(st.c)
```

DENSITY就是所谓浓度，也就是使用这个条件式判断哈希表是否已经趋于拥挤。st_table中如果所在同一个bin_pos的值过多的话，链表就会变成，搜索速度就会变慢。所以当元素个数过多的时候，我们增加bin的大小来缓解这种拥挤。

现在所设定的ST_DEFAULT_MAX_DENSITY如下

```
  23  #define ST_DEFAULT_MAX_DENSITY 5

(st.c)
```

这个浓度被设置成了5，也就是说当所有的bin_pos链表的元素st_table_entry都已经达到5个的情况下，我们就要增大容量。

st_insert()

st_insert()不过是st_add_direct()和st_lookup()的组合而已。只要了解后两者就ok了。

```
 286  int
 287  st_insert(table, key, value)
 288      register st_table *table;
 289      register char *key;
 290      char *value;
 291  {
 292      unsigned int hash_val, bin_pos;
 293      register st_table_entry *ptr;
 294
 295      hash_val = do_hash(key, table);
 296      FIND_ENTRY(table, ptr, hash_val, bin_pos);
 297
 298      if (ptr == 0) {
 299          ADD_DIRECT(table, key, value, hash_val, bin_pos);
 300          return 0;
 301      }
 302      else {
 303          ptr->record = value;
 304          return 1;
 305      }
 306  }

(st.c)
```

首先查询元素是否已经被添加到哈希表中，如果还没有，就向哈希表中添加元素，实际上添加了元素则返回眞，如果没有添加就返回假。

ID和符号

我们已经说明过ID是什么东西。ID是和任意的字符串一一对应的数值，可以表示各种名称。实际的ID的类型是 unsigned int。

从char到ID

字符串到ID的变化通过rb_intern()进行。这个函数略长我们省略其中一部分。

rb_intern()（缩减版）

```
5451  static st_table *sym_tbl;       /*  char* → ID   */
5452  static st_table *sym_rev_tbl;   /*  ID → char*   */

5469  ID
5470  rb_intern(name)
5471      const char *name;
5472  {
5473      const char *m = name;
5474      ID id;
5475      int last;
5476
          /* 既にnameに対応するIDが登録されていたらそれを返す */
5477      if (st_lookup(sym_tbl, name, &id))
5478          return id;

          /* 省略……新しいIDを作る */

          /* nameとIDの関連を登録する */
5538    id_regist:
5539      name = strdup(name);
5540      st_add_direct(sym_tbl, name, id);
5541      st_add_direct(sym_rev_tbl, id, name);
5542      return id;
5543  }

(parse.y)
```

字符串和ID的一一对应使用st_table来实现。应该不是什么难点。

要说我们省略了什么，那就是当遇到全局变量或者实例变量的时候我们会进行特殊处理插入标记位。因为ruby的语法解析器需要从ID获取变量的类型。但是这些又和ID的原理没多大联系所以这里就不贴出来了。

从ID到char*

rb_intern()的逆向，从ID获取char*使用的是rb_id2name()这个函数。大家应该已经明白id2name的2是to的意思。因为to和two发音相同所以被替代使用了。这个写法出乎意料相当常见。

这个函数也是根据ID的种类设立各种flag标记，所以变得很长。我们尽量删掉无关紧要的部分来看这个函数。

rb_id2name(阉割版)

```
char *
rb_id2name(id)
    ID id;
{
    char *name;

    if (st_lookup(sym_rev_tbl, id, &name))
        return name;
    return 0;
}
```

是不是觉得有些过于简单了。其实只是删除掉了一些小细节。

这里需要注意，我们没有拷贝需要检索的name。Ruby的API的返回值不需要free()(绝对不能free)。另外传递参数的时候通常会通过拷贝来使用。也就是说生成和释放通常是用户或者ruby的一方来执行完成的。(谁生成谁释放)

那么对于生成和释放无法相对应的值(一旦被传递就无法控制)是如何处理的呢？这个时候会要求使用Ruby对象。Ruby对象会在我们不需要的时候自动释放。

VALUE和ID的互相转换

ID在Ruby层面上是Symbol类的实例, `"string".intern` 会返回对应的ID。这个String#intern的实体就是rb_str_intern()。

▼rb_str_intern()

```
2996  static VALUE
2997  rb_str_intern(str)
2998      VALUE str;
2999  {
3000      ID id;
3001
3002      if (!RSTRING(str)->ptr || RSTRING(str)->len == 0) {
3003          rb_raise(rb_eArgError, "interning empty string");
3004      }
3005      if (strlen(RSTRING(str)->ptr) != RSTRING(str)->len)
3006          rb_raise(rb_eArgError, "string contains `\\0'");
3007      id = rb_intern(RSTRING(str)->ptr);
3008      return ID2SYM(id);
3009  }

(string.c)
```

这个函数作为ruby的类库的代码例子来说真是信手拈来。注意其中使用RSTRING()强制转换来访问构造体成员的技巧。

读读代码吧。首先rb_raise()只是出错处理所以先无视。这个函数里面有刚才刚解释过的rb_intern(),ID2SYM().ID2SYM()是将ID转换成Symbol的宏。

这个操作的逆向是Symbol#to_s。其实体函数为sym_to_s。

▼sym_to_s()

```
522  static VALUE
523  sym_to_s(sym)
524      VALUE sym;
525  {
526      return rb_str_new2(rb_id2name(SYM2ID(sym)));
527  }

(object.c)
```

SYM2ID是将Symbol(VALUE)转换成ID的的宏。

看上去很不常见的写法，应该注意的是内存处理相关的地方。rb_id2name()返回一个不允许free()的char, *rb_str_new2()*将参数*char*拷贝使用(不会改变参数)。因为贯彻了这个方针所以可以写成函数套函数的连锁形式。

(第三章完)